Balancing Methods for Binary Search Trees

James Michael Cannady (N.C. State)

Binary search trees have received a great deal of attention in recent years. As a result of this interest, several methods have been developed for balancing them; namely, random, height-balanced, bounded-balance, and weight-balanced trees. These methods which include weighted and non-weighted binary search trees are grouped into two classes: 1) dynamic balancing and 2) total restructuring. The rational and properties of the more significant methods are discussed and compared with other tree balancing algorithms. These comparisons provide insight about the conditions under which an algorithm is appropriate.

1. Introduction

Binary search trees (abbreviated as B.S. trees) are important structures in organizing large files. An optimum B.S. tree allows faster random access compared to linked linear lists. A B.S. tree (optimum or otherwise) allows faster sequential access compared to scatter storage techniques. Although the configuration (shape) of a B.S. tree does not affect the performance of sequential access much, it can have a drastic affect on random access, the worst case being the same as a linked linear list. If there are frequent searches, one would want to reduce the search time in order to allow more searches done in an interval of time. This paper surveys existing methods used to keep the searching time of a B.S. tree near the optimum.

Section 2 covers the basic definitions used in conjunction with B.S. trees. Section 3 deals with some of the properties of B.S. trees that do not worry about the configuration, commonly called random B.S. trees. The volatility of the file represented by a B.S. tree affects the decision of either to optimize the B.S. tree periodically if the file is static or to optimize the tree continuously. The first is covered in section 4 called total restructuring, the second is covered in section 5 called dynamic restructuring.

2. Basic Definitions

A Binary Search Tree is a finite set of nodes either empty (called a null tree) or the triple T (T_1, r, T_R) where r is the root of the tree and T_L and T_R are, respectively, the left and right subtrees of r. Each node S, except the null node, is called the parent of its left and right subtree roots (designated T_L^S and T_R^S). The left and right subtrees are themselves binary search trees. Each node, except the empty node, contains 4 items:

1. The key of the node which identifies the node
2. data area, which may be empty
3. A left link, to the root of the left subtree. This is null if the subtree is empty.
4. A right link to the root of the right subtree. This is null if the subtree is empty.

A B.S. tree also has the following property: for all non empty nodes $S \in T$,

$V_{S_i} \in T_L^S$, Key (S_i) < Key (S) and

$V_{S_j} \in T_R^S$, Key (S_j) > Key (S).

This forbids equal keys. Figure 1a is a B.S. tree but figure 1b is not a B.S. tree because 'B' ≠ 'C'.

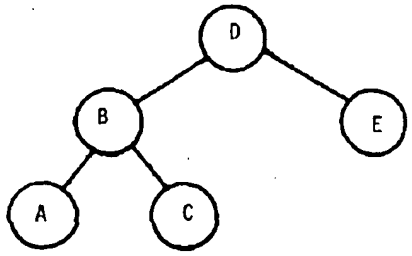

Figure 1a. A Binary Search Tree

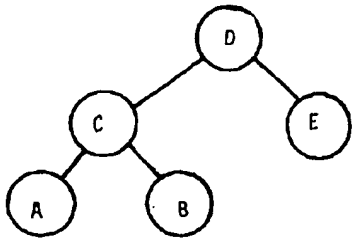

Figure 1b. Not a Binary Search Tree (C ≠ B)

The depth or path length of a node S in tree T is 1 if S is the root node of T else it is 1 plus the path length of the parent of S. For all nodes S ε tree T, the (Total) path length of tree T is Σ depth (S). The height of tree T is 1 plus the maximum of the heights of T_L and T_R, if T is not a null tree, else it is 0.

A weighted B.S. tree is a B.S. tree in which there is associated a weight for each node as contrasted to non-weighted trees which have no weights associated with each node. One use for the weights is to indicate the frequencies that each node is accessed. Balancing a weighted tree should take into account the weights so that the larger ones will be closer to the top of the tree and the lessor ones be nearer the bottom. The overall effect is to reduce the total weighted path length of the tree.

For all nodes S in the weighted B.S. tree T the total weighted path length is defined to be Σ(Weight(S)*path length(S)). The basic searching algorithm for a node with Key S is basicly this:

1. Compare S with the Key in the node.
1.1 If S = Key then the node is found.
1.2 If S < Key, consider the left subtree of the current node.
1.3 If S > Key consider the right subtree of the current node.
1.4 If the subtree to be considered next is null, the node is not in the tree.

The basic insertion algorithm for a node with Key S is the same as searching except:

1. If in 1.1 the keys are equal the node is already in the tree.
2. The insertion is done when the null tree is encountered at 1.4.

The basic deletion algorithm for a node with Key S is to:

1. Find the node with its key = S.
2. Find the lexicographical successor after S that is in the right subtree of S.
3. Delete the successor and place it where node S is.

3. Randon Binary Search Trees

Random B.S. trees are B.S. trees in which there is no effort made to balance or optimize the tree. There is also the assumption that the keys used in the operations on the tree are random.

It is self-evident that insertion of random keys into a null or random B.S. tree remains a random B.S. tree. Knuth [2] shows that after a random element is deleted from a random B.S. tree, the resulting tree is still random. He also states that a combination of insertions and deletions (e.g. insert, delete, insert, delete, etc) destroys the randomness of the B.S. tree and that the path length tends to decrease. If one has mostly insertions and deletions in comparison to searches and there is a "good" balance of insertions and deletions then one could use random trees to take advantage of this tendency of reducing the path length of the tree. As of yet, there is no theoretical explanation for this behavior.

Martin and Ness [1] point out that for non-weighted trees grown without any reorganization were quite good on the average and that reorganization was a second order improvement.

Random B.S. trees may be practical if

1. The frequency of access is very low and does not justify even very lose balancing.
2. The tree is to have only a "few" nodes and/or
3. There is a good ratio of insertions and deletions in comparison to searches.

Random B.S. trees may not be practical if

1. There is a high frequency of access and access time is critical.
2. There are many nodes in the tree
3. There are quite a bit more insertions than deletions
4. Mainly searchers are done, thus total restructuring may be more appropriate.

In these guidelines, one must remember that these are overall guidelines and may not be suitable to a tree built with non-random keys. Whether this method is good for those applications can only be done by testing. For example if the application has quite a few (or all) insertions this method is probably not applicable, it depends on the sequence of keys. Whereas, if a combination of insertions and deletions are allowed it might be applicable, again it depends on the sequence of keys in deleting and inserting.

4. Total Restructuring

Total restructuring attempts to build optimum or near optimum B.S. trees by completely rebuilding the tree. This method is good for very static files in that balancing is not done continuously but periodically and can be done in conjunction with end-of-day, end-of-week, etc., processing or when it is determined to reorganize. Generally, one can not search the tree when total restructuring is going on.

4.1 Non-weighted Trees

For non-weighted B.S. trees, Martin and Ness [1] gave a simple algorithm which build optimum B.S. trees. It requires the knowledge of a number of nodes to be in the tree and must access the keys in the old tree sequentially. It generally does the following:

1. Calculate the number of nodes for the left and right subtrees.
2. Builds the left subtree by calling itself with the left subtree node count.
3. Builds the root node.
4. Builds the right subtree by calling itself with the right subtree node count.

The algorithm requires only O(N) time and $O(\log_2 N)$ storage for the stack, where N is the number of nodes of the old tree.

Table 1 New Balance Factors after a rotation
BF1 = Balance factor before rotation
BF2 = Balance factor after rotation

1. Right Single Rotation (Figure 2a):
   BF2(A) = -K + max (Ø, -BF1(B))
   BF2(B) = BF1(B) +1
2. Left Single Rotation (Figure 2b):
   BF2(A) = K - max (Ø, BF1(B))
   BF2(B) = BF1(B) - 1
3. Left Double Rotation (Figure 2c):
   BF2(A) = K - max (Ø, BF1(C)) -1
   BF2(B) = -K + max (Ø, - BF1(C)) +1
   BF2(C) = BF1(C) - sign (BF1(C))*Floor (ABS(BF1(C))/K)
4. Right Double Rotation (Figure 2d):
   BF2(A) = -K + max (Ø, -BF1(C)) + 1
   BF2(B) = K - max (Ø, BF1(C)) - 1
   BF2(C) = BF1(C) + sign (BF1(C))*Floor (ABS(BF1(C))/K)

5.1.2 Bounded balance trees

A B.S. tree is of bounded balance designated BB(K) for $0 \leq K \leq 1/2$ if and only if either $N \leq 1$ or for $N > 1$ the following holds for each node J
1. $K \leq (L+1) / (N+1) \leq 1 - K$ and
2. both subtrees of the node are of BB(K), where N is the number of nodes in the tree whose root is J and L is the number of nodes in the left subtree of node J. The root balance of a node is (L+1) / (N+1) for that node.

This class of B.S. trees was introduced by Nievergelt and Reingold [7]. As the definition indicates ,this class only considers the number of nodes in the tree (and subtrees). A completely balanced and full tree would be BB(1/2); the Fibonacci trees would be BB(1/3); random trees would be BB(0).

The requirements for using this is the knowledge of the number of nodes. One way to do this is to store a datum in each node indicating this. One good point of this method in that one can forcast what the root balance of a node is when the insertion or deletion is done, thus allowing one to do restructuring while going down the tree, although if the insertion or deletion fails, one would have to go through the tree again to possibly undo what was done. Once a critical node is identified, the rotation type and direction has to be determined and the rotation is done. One problem with this is that for small trees (e.g. has 2 nodes) the rotation could be quite drastic. This situation should be a special case. Another problem is that inserting and deleting can cause multiple restructurings.

Once a critical node is identified, one has to decide on a single or double rotation. Since trees of $BB(1- \sqrt{2}/2) = BB(1/2)$ [7] and if one restricts $K \leq 1 - \sqrt{2}/2$ then the following decides the rotation type:
1. Let threshold = (1-2K)(1-K)
2. If the direction is to the left then calculate what the root balance of the right subtree would be after insertion or deletion. If this is less than threshold do a single rotation else do a double rotation.
3. If the direction of the rotation is to the right then calculate what the root balance of the left subtree would be after the insertion or deletion. IF this is greater than -(Threshold + 1) then do a single rotation else do a double rotation.

One criticism of this method is that to for imbalance one would have to divide in calculating the root balance. Using the basic definition, a node is balanced if the following conditions are true:
1. $(L + 1) \geq K(N + 1)$ and
2. $(N + 1) - (L + 1) \geq K (N + 1)$ or $R \geq K(N + 1)$, where R is number of nodes in the right subtree.

This can be used to determine the direction of the rotation although it does not help the root node calculation in determining a double or single rotation. It seems that one could make this test more symetrical by defining the conditions as:
1. $L \geq K (N - 1)$ and
2. $R \geq K (N - 1)$ or $(N-1) - L \geq K(N-1)$,
Thus making Part 1 of the definition to be:

$$K \leq L/(N-1) \leq 1 - K$$

The theorms by Nievergelt [7] may not be applicable in this case and more work needs to be done in this case.

The K factor in this case is similar to the K factor in H.B. trees. It allows how tight or lose the trees is balanced. It can be varied at anytime without problems occurring.
The advantages of B.B. trees are:
1. Allows rebalancing as one goes down the tree.
2. K can vary in rather small increments since it is a real number (opposed to an integer).
3. Allows easier implementation of finding the Jth node in the lexicographical ordering of the tree since the node counts of a subtree is in the subtrees root node.

The disadvantages of B.B. trees are:
1. If an insertion or deletion fails one would have to undo what was done.
2. The node count datum has to be big enough to store the largest number of nodes allowed in the tree.

Insertion and deletion are clearly depended upon search time of a B.B. tree. The maximum height of a BB(K) tree is $(\log_2 (N+1) - 1)/ \log_2 (1/(1 - K))$ [7]. The total path length (T) is calculated as:

$$T=[(N+1) \log_2 (N+1)-2N]/[-K \log_2 K-(1-K)\log_2(1-K)]$$

Thus the average path length is T/N.

The average path length give the average search time and the maximum height gives the worst search time.

5.1.3. Height balanced vs. Bounded balanced trees

The differences between HB and BB trees are:
1. The K in H.B. trees has to be an integer and a change in this can change the characteristics of the tree considerably; Whereas in BB trees the K is a real number and can be changed minutely.
2. If the K in H.B. trees is decreased, it can lead to problems in the balancing method but with BB trees a change does not bother the method.
3. H.B. trees can not be described as BB trees always and BB trees can not be described as H.B. trees always.

5.2. Weighted Trees

5.2.1 Weighted-balanced Trees

Baer [8] introduced a dynamic balancing heuristic which locally balances a weighted B.S. tree designated W.B. trees. It is based on the definition of a trees weighted path length (WPL) which can be stated as :
WPL(T) = Total (T) + WPL(left subtree of T) + WPL(right subtree of T)
Where total is the sum of the weights of all the nodes in tree T.

This method can be used top-down or bottom-up or both. For example if one is inserting and knows the weight of the node being inserted one can test for imbalance and do the rotations as one goes down the tree. If the node is there already one can then backtrack and do rebalancing if needed. Likewise if one is deleting, one can go down the tree, delete the node and rebalance on the way back. One can also do balancing while searching the tree.(e.g. The weights are access counts of the nodes and searching alters these counts.)

The requirements of this method is that the total of the weights of the tree (or subtree) be stored in the tree's (or subtree's) root node along with the nodes weight.
When balancing the tree (when going down or up) The following tests and rotations are done:
1. A right single rotation is done (figure 2a) if the following is true before the rotation:
weight (B) + Total (left subtree of B) > weight (A) + Total (right subtree of A)
2. A Right double rotation is done (figure 2d) if a single rotation is not applicable and the following is true before the rotation:
weight (C) + Total (C) > weight (A) + Total (right subtree of A)

The left rotations are symetrical to the right rotations.

One can use the WB method to balance non-weighted trees by allowing the weights of all the nodes to be 1. If this is the case then this method builds trees that are BB(1/4) [8], thus the maximum height is $O(\log_2 N)$.

Baer also extended his method by introducing a parameter (K) to relax the balancing criterior. The test would then be for tests 1,
Weight (B)+Total(Left subtree of B) > Weight (A)+Total(Right subtree of A)+K and for 2,
Weight(C)+Total(C)>Weight(A)+Total(right subtree of A)+K

This K factor allows find tuning of the tree and can be changed in either direction and should not cause problems. The advantages of WB trees are:
1. The weights of the nodes can change and be balanced when searching
2. The weights of nodes are taken into account when balancing.

The disadvantages are:
1. Quite a few rebalancing can occur(depending on the weights).

6. Summary

This paper presents major balancing methods currently in the literature with general guidelines of the applicability of each as there is no absolute rule to determine which is better. The literature lacks empirical data about the effects of a combination of insertions and deletions on the average search time of the different methods; it has dealt with insertions as a rule. Baer [8] [9] shows the average search times and number of rotations per insertion and discusses the costs of the different dynamic methods.

References

1. Martin, W.A. and Ness, D.N. "Optimizing Binary Trees Grown with a Sorting Algorithm." Comm. ACM, 15,2 (Feb. 1972), 88-93

2. Knuth, D.E. The Art of Computer Programming, Vol. 3, Sorting and Searching. Addison-Wesley, Reading, Mass. 1975.

3. Nievergelt, J. "Binary Search Trees and File Organization", Computing Surveys 6,3 (Sep. 1974) 195-207.

4. Adel'son - Vel'skiy, G.M., and Landis, E.M. "An Algorithm for the Organization of Information." Doklady, Akad. Nauk. USSR 146, 2(1962) 263-266.

5. Foster, C.C. "A Generalization of AVL Trees." Comm. ACM 16, 8(Aug. 1973), 513-517.

6. Karlton, P.L. et al. "Performance of Height-balanced Trees." Comm. ACM 19, 1(Jan. 1976) 23-28.

7. Nievergelt, J. and Reingold, F.M. "Binary Search Trees of Bounded Balance." SIAM J. Computing 2, 1(March 1973) 33-43.

8. Baer, J.L. "Weight-balanced Trees." Proc. AFIPS 1975 NCC, Vol. 44, AFIPS Press, Montvale, N.J., pp. 467-472.

9. Baer, J.L. and Schwab, B. "A Comparison of Tree-balancing Algorthms." Comm. ACM 20, 5 (May 1977) 322-330.